Compilation of Structured Polyhedral Equations

Yun Zou, Guillaume Iooss, and Sanjay Rajopadhye
Computer Science Department
Colorado State University, Ft. Collins, CO
zou_yun@yahoo.com, guillaume.iooss@gmail.com, Sanjay.Rajopadhye@colostate.edu

Abstract—The polyhedral model is an established mathematical formalism for automatic parallelization of an important class of programs. In 1989, Mauras defined AL.PHA, a polyhedral equational language based on systems of affine recurrence equations over polyhedral domains. In 1995, Dupont de Dinechin introduced subsystems in AL.PHA to allow modularity and structured programming. Such hierarchical structure is now regaining importance when polyhedral programs are tiled with parametric tile sizes. Such tiling transformations are non-linear, and render the program representation “out of the scope” of polyhedral representation. Current approaches apply them at the final, code-generation step. Structured polyhedral programs offer an alternative solution, the ability to work with a polyhedral specification of programs with parametric tile sizes. This paper addresses the issue of code generation and compilation of such structured AL.PHA programs. We implement this in our polyhedral transformation framework (AlphaZ). We also give examples to illustrate the ability to express the complex programs and to demonstrate the code generation.

I. INTRODUCTION

The polyhedral model is a powerful framework for automatic parallelization and program optimization. It provides a mathematical abstraction to reason about a class of imperative programs, called affine control loop programs (the inaccurate term SCoP or static control parts is popular, but incorrect—an FFT program has static but non-affine control, and cannot be analyzed by polyhedral methods). It enables instance-wise analysis of statements in a program (since statements may be surrounded by loops and therefore have multiple instances) and element-wise analysis of arrays (rather than treating all arrays as single monolithic entities). However, underlying the program model is a declarative formalism called systems of affine recurrence equations.

AL.PHA [1] is a strongly typed, functional polyhedral language, originally defined by Mauras [1] in 1989, and extended by Le Verge [2] to include reductions as first class expressions. AL.PHAZ [3] is a tool developed at CSU that allows the manipulation, analysis and transformation of the AL.PHA programs. It is similar in structure to an earlier tool MMALPHA [4] developed at IRISA, Rennes, France. MMALPHA targeted VLSI processor arrays and generated code in the form of a hardware description language, while AL.PHAZ targets modern multi- and many-core processors, and generates parallel code (C+OpenMP and C+MPI).

One key limitation of MMALPHA was that it did not handle tiling, which was then, and still is, a difficult problem. However, tiling is a critical program transformation and is used extensively to address different aspects of performance, ranging from data locality to granularity of parallelism, and control of communication-computation balance. Parametric tiling—tiling when tile sizes are parameters to be instantiated later in the compilation process, or even at run time—is a known limitation of polyhedral model. Most current polyhedral tools do not handle parametric tiling cleanly. The fundamental difficulty is that tiling with parametric tile sizes is a non-linear transformation, and therefore violates the closure properties necessary for program transformations. As a result, the “workaround” to the problem is to do parametric tiling only at the final, code-generation step when there is no need for further polyhedral analyses and transformations. All the state-of-the-art polyhedral tools adopt this strategy [5], [3] or limit themselves to fixed size tiling [6], [7], [8].

However, this is not very satisfactory, since the representation of a tiled program is no longer polyhedral, and therefore cannot be subject to further polyhedral transformation or analysis. For example, the output produced by most current polyhedral compilers cannot be fed back directly into a polyhedral compilation/optimization system.

One alternative is to perform the necessary non-linear transformations before any polyhedral analysis and transformations, and work with a purely polyhedral representation. At the expense of an initial, possibly human-guided, and non-obvious preprocessing, one has a polyhedral representation throughout the transformation and code generation process. This motivates the need for tools and systems to manipulate hierarchical polyhedral programs—polyhedral programs in which tiling has been performed at the source/algorithmic level.

In 1995, Dupont de Dinechin [9], [10] argued for the need to include modularity in AL.PHA, and proposed the notion of subsystems in the language. He provided a semantics for such programs and presented a number of basic analyses and transformations. The use of subsystems was limited to structural description of regular hardware in MMALPHA, notably in the AlpHard language [11]. The need for tiling, and in particular, the idea of performing tiling as a first, non-linear preprocessing transformation, motivates us to revisit subsystems in AL.PHA, and is the topic of the current paper. Our main contributions are as follows:

- We revisit the definition and foundations of subsystems

1When tile sizes a compile-time constants, tiling can be described as an affine transformation and can be handled within the polyhedral model.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Syntax</th>
<th>Domain</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>Constant symbol</td>
<td>$\mathbb{Z}^0$</td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Variable name $X$</td>
<td>$\mathcal{D}_X$</td>
<td>Declared domain</td>
</tr>
<tr>
<td>Operator</td>
<td>$\text{op}(\text{Exp}_1, \ldots, \text{Exp}_M)$</td>
<td>$\bigcap_{i=1}^M \mathcal{D}_{\text{Exp}_i}$</td>
<td>infix syntax too</td>
</tr>
<tr>
<td>Case</td>
<td>case $\text{Exp}_1; \ldots; \text{Exp}_M$ esac</td>
<td>$\bigcup_{i=1}^M \mathcal{D}_{\text{Exp}_i}$</td>
<td>disjoint union</td>
</tr>
<tr>
<td>Restriction</td>
<td>$\mathcal{D} : \text{Exp}$</td>
<td>$\mathcal{D} \cap \mathcal{D}_{\text{Exp}}$</td>
<td>relational inverse</td>
</tr>
<tr>
<td>Dependence</td>
<td>$f@\text{Exp}$</td>
<td>$f^{-1}(\mathcal{D}_{\text{Exp}})$</td>
<td></td>
</tr>
<tr>
<td>Reduction</td>
<td>reduce($\oplus$, $f$, Exp)</td>
<td>$f(\mathcal{D}_{\text{Exp}})$</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. ALPHA expressions and computation rules for the associated domains. A case expression describes a conditional expression defined over disjoint domains. A restriction expression just restricts the definition domain of its sub-expression. The value of a dependence expression $f@\text{Exp}$ at an index point $i$ is the value of $\text{Exp}_i$ at the point $f(i)$. In the case where the sub-expression is a variable $X$, the value of $f@X$ at the point $i$ can be written as $f[X(i)]$. A reduction expression corresponds to an accumulation with the operator $\oplus$. The values of $\text{Exp}$ at all points that are mapped by $f$ to the same index point $i$ are accumulated together, using the associative and commutative operator $\oplus$.

in ALPHA as proposed by Dupont de Dinechin [9], [10] and propose new static analyses and transformations. In particular, we extend a critical program transformation called Change-of-Basis to operate on ALPHA programs with subsystems. We also implement in ALPHAZ, some known transformations and analysis that were not previously implemented in MMALPHA.

- We describe a methodology for compiling ALPHA programs with subsystems to parallel, shared memory code (sequential code is a special case) in OpenMP, and implement a code generator embodying this methodology. Besides the correctness of the generated code, our code generator is able to generate code incorporating a number of optimizations such as memory reuse and value reuse.

II. BACKGROUND

In this section, we briefly introduce ALPHA\(^2\), including Dupont de Dinechin’s extension to subsystems.

A. The ALPHA Language

An ALPHA program is a collection of affine systems, each affine system consists of:

- A parameter domain, which is a polyhedral domain that specifies the constraints on the program parameters.
- A list of declarations of input/output/local variables $\text{Var}$, each associated with a polyhedral domain $\mathcal{D}_{\text{Var}}$.
- A list of equations $\text{Var} = \text{Expr}$. There is exactly one equation for each local or output variable. The syntax of expressions is given in Fig 1.

The analysis phase computes two domains for each sub-expression:

\(^2\)A detailed presentation of the formalism can be found in [12]

- Expression domain, which is the set of points in which the expression is defined.
- Context domain, which is a subset of the expression domain, and is the set of points at which the expression needs to be evaluated, in order to compute the outputs of the system.

The expression domain is computed recursively, in a bottom-up fashion, using the rules in the last column of Fig 1. The context domain is computed in a top-down fashion.

*Example:* Consider the equation $A.x = b$ where $A$ is a lower triangular matrix whose diagonal values are 1 and $x, b$ are vectors. Given $A$ and $b$, forward substitution solves this equation for $x$ using the formula, $x_i = b_i - \sum_{j=0}^{i-1} A_{i,j}x_j$, with appropriate boundary cases. An ALPHA program is shown in Fig 2.

B. Subsystems

Dupont de Dinechin [9], [10] introduced the notion of modularity in ALPHA. A subsystem is an affine system which is “called” or invoked from another system on a specific set of parameters and inputs values. Indeed, a “polyhedral collection” of systems may be invoked in a compact manner. Fig 3 shows a modular version of the forward substitution example above, where each instance of the reduction is written as the invocation of a dot-product subsystem.

The equation describing this system call is called a use equation and has the following syntax:

```
use $\mathcal{D}_{\text{ext}}$ $s[p]$ (i) (o);
```

where:

- $\mathcal{D}_{\text{ext}}$ is the [optional] extension domain.
- $s$ is the name of the subsystem called.
III. STATIC ANALYSIS OF SUBSYSTEMS

We first present our adaptation of the context domain computation to subsystems. Then we describe the Change-of-basis transformation on use equations and explain how to extend a data structure called the polyhedral reduced dependence graph (PRDG) to use equations.

A. Context domain for the input expressions of a use equation

For a standard ALPHA equation \( \text{Var} = \text{Expr} \), the context domain of \( \text{Expr} \) is exactly the definition domain of \( \text{Var} \). Then, we recursively compute the context domain of each subexpression of \( \text{Expr} \). However, the input expressions of a use equation are not included inside any standard ALPHA equation, and therefore we need to introduce new rules to handle this.

Dupont de Dinechin introduced an inlining transformation for subsystem [9, Section 4.5]. This inlining transformation creates a new local variable in the main system for each input/output variable of the inlined subsystem, and all the equations in the subsystem is copied into the main system with these new variables. Moreover, every domain or function from the subsystem is modified, to account for the extension domain and the parameter changes. Also, a copy equation is added for every input of the subsystem, and has the form \( \text{Var}_{in} = \text{Expr} \), for the input equations.

To determine the set of points of an input expression which will be used by a subsystem, we can translate the domain of the corresponding input variable in the subsystem, using the transformation introduced by Dupont de Dinechin for his inlining transformation. This is equivalent to consider the corresponding input copy equation, then apply the context domain computation on this standard equation.

B. Change of Basis on a use equation

The Change of Basis (CoB) is a very useful transformation which subsumes many others (such as loop skewing, interchanging) in a single formalism. It consists of transforming the domain \( D_{\text{Var}} \) of a variable into \( f(D_{\text{Var}}) \), where \( f \) is a bijective affine function. This transformation impacts the affine system in the following way.

- The equation \( \text{Var} = \text{Expr} \) is transformed into \( \text{Var} = f^{-1}@\text{Expr} \)
- Each occurrence of \( \text{Var} \) is replaced by \( f@\text{Var} \)

The CoB transformation cannot be applied directly to an output variable \( \text{Var}_{out} \) of a use equation. Indeed, if \( f \) touches the dimensions of the extension domain, it modifies the
domains of all the outputs of the use equation. The ALPHAZ system returns an error and advises the user to introduce a copy variable. However, the user can force the transformation, in this case, we perform the same CoB to every output variable of the use equation.

It is also possible to apply a CoB directly on a use equation: the idea is to transform its extension domain \( D_{ext} \) into \( f(D_{ext}) \) where \( f \) is bijective. This transformation will impact the affine system in the following ways:

- The extension domain \( D_{ext} \) is replaced by \( f(D_{ext}) \)
- Each input expression \( Expr \) is replaced by \( (f^{-1} \times Id_{k})@Expr \) where \( k \) is the dimension of the input of the subsystem.
- We perform partially a CoB on each output variable, using the function \( (f \times Id_{k'}) \) where \( k' \) is the dimension of the output of the subsystem.

This transformation is used to permute or skew a loop nest, to enable other transformations (such as tiling) or to expose parallelism.

C. Outlining transformation

The outlining transformation consists of extracting a set of equations from a main system, and creating a subsystem from them. The input variables of this new subsystem are the variables used in these equations and defined elsewhere. The output variables are the variables defined in these equations and used elsewhere. The local variables are the variables defined in these equations and used only inside these equations.

Our current version is simple: the use equation do not have any extension domain, and input expressions to the use equation can be specified by the user. Eventually, we will allow the user to define an extension domain, which will impact the domain of the variables of the subsystem and the use equation created.

D. Polyhedral Reduced Dependence Graph

To analyze statically the dependencies of a program, we have to create a Polyhedral Reduced Dependence Graph (PRDG). The nodes in a PRDG are the variables of a program. There is an edge between node \( a \) (source) to node \( b \) (destination) if the computation for variable \( a \) depends on variable \( b \). The nodes and edges are labeled respectively by the domain of the variable and the dependence function.

We chose to represent a use equation in a PRDG in the following way:

- A new node is created for each input expression
- The use equation is represented by a hyper-edge, whose sources are the previously created nodes for input expressions, and destinations are the nodes for the output variables.

Moreover, we label the hyper-edge by the name of the subsystem called, the parameters and the extension domain of the use equation.

IV. CODE GENERATION

Code generation is an important component in ALPHAZ, and several code generators are available [3]. The most sophisticated one ScheduledC, generates C++-OpenMP code. We extended this code generator to support code generation for subsystems. In this section, we will describe the strategy for the code generator and illustrate how to use it to generate efficient code.

In ALPHAZ, compilation is based on three orthogonal components.

- First, a target mapping specifies, for each variable in the ALPHA program, all the aspects of the desired parallelization. This includes the schedule, the memory allocation, and tiling—which dimensions to tile, how many levels of tiling, and the tile sizes at each level.
- Next, a verifier checks the legality of the proposed mapping (this is because the target mapping may optionally be specified interactively by the user).
- Finally, a code generator in the form of a very sophisticated “pretty printer” based on CLooG [13] that actually produces the target code.

The choice of the target mapping is a long standing research problem, since the target architectures are continually evolving. Therefore ALPHAZ provides the ability for users to explore these choices manually.

A. Target Mapping

The main components of the target mapping are space-time map and memory map. The space-time map has two components, a schedule and a processor allocation, thus it determines when and on which virtual processor the computation should be executed. A space-time map is specified for each variable, and is a bijective multi-dimensional affine function [14], each dimension can be sequential, parallel or tagged as an “ordering” dimension. For example, we can give a space-time map \( (i \rightarrow i, 0) \) to the equation \( x \) in the forward substitution example. This tells the code generator that the statement generated for the \( i \)th index of \( x \) is to be computed at time step \((i,0)\), and the second dimension is an ordering dimension. The memory map, represents, for every variable in the program, its memory allocation, and is also specified by a multi-dimensional affine function, possibly annotated with modulo functions. For example, we can set the memory map for the variable \( temp \) in the forward substitution to be \((i,j \rightarrow i)\), which means the \((i,j)\)th value is stored at the memory location \( i \).

To generate codes with parallelized C loops, the target mapping also needs to have additional information. The parallelization specification tells the code generator which loops are to be annotated with \texttt{pragma omp parallel}. Since specific loops may not correspond directly to the dimensions
of an index, and so, this is specified as the $n$th dimension with a certain ordering prefix in the space-time map. For example, given the space-time map $\{i \rightarrow i, 0\}$ for statement $x$, we can parallelize the $i$ dimension with the specification: the first dimension with empty ordering prefix.

In compiling ALPHA programs with subsystems, the main assumption is that instances of subsystems are atomic, i.e., they can be executed with the following protocol: first evaluate (and collect) all the inputs to (an instance of) the subsystem, then evaluate all the local and output variables of that instance, and finally, retrieve the outputs and continue execution of other computations in the main system. Atomicity can be verified by a simple extensions to the algorithm that determines the subsystem schedules [15].

B. Strategy

We now describe our strategy for subsystem code generation, and the necessary extensions to target mapping.

Because of atomicity of subsystems, we can implement them as function calls in C. Where the function call statement is placed is decided by the space-time maps in target mapping. A space-time map is specified for every use equation and its left hand side has the same number of dimensions as $D_{ext}$. In order to implement function calls, the key problem is how to pass the correct values as inputs or store the outputs into the correct place. Simply passing some pointer of the existing variable is incorrect for two reasons. First, the inputs of subsystems can be an expression and not necessarily a variable, so there may not be any memory allocated for it in the calling system. Second, the set of indices corresponding to an instance of a subsystem may be an arbitrary “slice” of the context domain of the expression, and hence, the appropriate set of values needs to be marshaled together into a contiguous region of memory.

Figure 4 shows the matrix multiplication using a `dot_product` subsystem: the $(i, j)$th value of the final matrix $C$ is computed by a dot product of the $i$th row of matrix $A$ and the $j$th column of matrix $B$ by the $(i, j)$th instance of the use. Although tiling is not addressed in this example, it is a simple example to illustrate the code generation strategy and possible optimizations. Later, we will show more examples where tiling is used.

For this program, if both matrix $A$ and $B$ are allocated in the standard row-major manner, the $i$th row of $A$ can be passed as $\&A[i]$, but the columns of $B$ are not in contiguous memory locations, but that is how a dot-product function expects its arguments. To generate correct codes, a temporary variable is created for every input/output of each use equation, that has the same memory size and allocation as the declared domains of the subsystem input/output, and corresponding values are copied to/from these variables before/after the function call.

To achieve this, our code generator generates three auxiliary statements for each input/output of each use equation: a memory allocation statement for each temporary variable, a set of value copy statements for the inputs and outputs, and the memory free statements for the temporary variables.

```c
// Product of rectangular matrices A and B
affine matrix_product_SubSyst {N,K,M | N>0 && K>0 && M > 0}
    input
    float A {i,k | 0<=i<N && 0<=k<K};
    float B {k,j | 0<=k<K && 0<=j<M};
    output
    float C {i,j | 0<=i<N && 0<=j<M};
    let
    use {iP,jP|0<=iP<N && 0<=jP<M}
    dot_product[K]
    ((pi,pj,k->pi,k)@A, (pi,pj,k->k,pj)@B)
    returns (C);

Fig. 4. Matrix multiplication using dot product subsystem
```

Figure 5 shows the code that is generated for the system `matrix_product_SubSyst`. Both of the loops in this example have been specified as parallel. Since parallelism just a simple annotation as far as the code generator is concerned, the remaining examples focus on sequential code.

By default, the temporary variables and special statements are created for every instance of the subsystem call, since the memory size and values that are needed for every subsystem instance may be different (e.g. in the forward substitution example of Figure 3, the $i$th instance of the dotproduct system operates on vectors of length $i - 1$). However, in other cases, reuse may be possible. For example, in the matrix
//declare temporary variable
long* UseEquation_C_input_0;
long* UseEquation_C_input_1;
long UseEquation_C_output_0;

//allocate memory for the inputs
UseEquation_C_input_0 = memory_allocation_for_UseEquation_C_input_0(K);
UseEquation_C_input_1 = memory_allocation_for_UseEquation_C_input_1(K);
for(i = 0; i <= N-1; i++){
    value_copy_for_UseEquation_C_input_0(K,i,j,A,UseEquation_C_input_0);
    for(j = 0; j <= M-1; j++){
        //value copy statements for inputs
        value_copy_for_UseEquation_C_input_1(K,i,j,B,UseEquation_C_input_1);

        //function call
        dot_product(K,UseEquation_C_input_0, UseEquation_C_input_1,
                    &UseEquation_C_output_0);

        //value copy statements for outputs
        value_copy_for_UseEquation_C_output_0(K,i,j,C, &UseEquation_C_output_0);
    }
}

//memory free
free(UseEquation_C_input_0);
free(UseEquation_C_input_1);

Fig. 6. The generated code for the matrix multiplication with memory and value reuse

For the matrix multiplication example, we can completely hoist the memory allocation statements for both inputs, and the value copy statement for the first input by one level (the copy of the i-th row of A is reused across the inner j loop). The code generated with these three optimizations is shown in Figure 6.

C. Legality check for target mapping

Our code generator accepts the target mapping as gospel and generates code that respects the given mapping. A separate module in the system is responsible for ensuring that the proposed target mapping is legal. For code generated without subsystems, this legality check is performed using existing polyhedral analyses [16]. We now explain how these need to be extended to handle the legality of target mappings for subsystems. Due to the space limitations, we only describe the legality checks for the sequential case.

In addition to the legality of the subsystem schedule including the test of atomicity, two additional checks need to be performed for the target mapping validation for subsystems: dependency check and reuse check. The dependency check validates whether the space-time map correctly specifies the loop hoisting optimization and whether all the dependencies in the program are respected. The reuse check checks whether the memory allocation and copy respect the lifetimes of the inputs.
values concerned. The dependency check has the following components:

1) By default, the special statements for the inputs/outputs of a subsystem are placed within the same loop with the subsystem call (as shown in Figure 5). This requires the space-time map for the special statements to have the same linear part as the space-time map for the use equation. Assume the space-time map for a special statement is \( \Theta_s \), the schedule for the use equation is \( \Theta_{use} \), and \( lin(\Theta) \) represents the linear part of the space-time map, then we have to check \( lin(\Theta_s) = lin(\Theta_{use}) \).

2) The dependencies between the special statements and the subsystem call have to be respected (these dependencies are in addition to those that were in the original program). For example, the value copy statement must happen after the malloc statement. Let \( \Theta_{malloc}, \Theta_{copy}, \Theta_{free}^{input}, \Theta_{malloc}^{input}, \Theta_{copy}^{input}, \Theta_{free}^{output}, \Theta_{malloc}^{output}, \Theta_{copy}^{output}, \Theta_{free}^{output} \) be the schedule for the memory allocation statement, value copy statement and memory free statement of the \( i \)th input. Similarly, we also have \( \Theta_{malloc}^{output}, \Theta_{copy}^{output}, \Theta_{free}^{output}, \Theta_{malloc}^{output}, \Theta_{copy}^{output}, \Theta_{free}^{output} \) for the \( i \)th output. Then the check we have to do is to ensure that \( \forall z \in \mathcal{D}_{ext} \),

\[
\Theta_{malloc}^{input}(z) < \Theta_{copy}^{input}(z) < \Theta_{use}(z) < \Theta_{free}^{input}(z) \\
\Theta_{malloc}^{output}(z) < \Theta_{copy}^{output}(z) < \Theta_{free}^{output}(z) \\
\Theta_{use}(z) < \Theta_{copy}^{output}(z)
\]

3) All other dependences in the program have to be respected by the space-time maps, and this is exactly what the ALPHAZ verifier already does for programs without subsystems [16]. In addition, the space-time map must be checked for atomicity of subsystems.

The reuse check checks whether the memory reuse and value reuse is legal. The \( z \)th instance of the subsystem call can share the same memory allocation statement with the \( z' \)th instance of the subsystem call for the \( i \)th input/output, if and only if the memory size for the \( i \)th input is the same for the two instances of the subsystem. When the memory allocation statement is hoisted out the \( m \)-nested loops for \( k \) levels, the subsystem calls with the common first \((m-k)\) indices all must have the same memory size for the \( i \)th input. Therefore, the vertices of domain for the \( i \)th input of the subsystem should not depend on any of the inner \( k \) loop indices. This is a simple check using existing polyhedral machinery.

For the value reuse, the \( z \)th instance of the subsystem call can share the same value copy statement with the \( z' \)th instance of the subsystem call for the \( i \)th input, if and only if the values needed by both instances are the same. A value copy statement can be hoisted out the innermost \( k \) loops if it has a reuse along the direction defined by \([0,1_k]\), the vector which is 1 in the innermost \( k \) dimensions and 0 in the outer ones. Again, we can use known polyhedral machinery [17] to perform these checks.

V. RESULTS

So far, we used simple examples to illustrate subsystems and our compilation strategy. Now, we demonstrate how to write structured tiled ALPHA code for the tiled forward substitution and tiled matrix multiplication, and also show the sample codes we generated with different optimization.

A. Tiled Matrix Multiplication

1) Alpha program for tiled matrix multiplication: In matrix multiplication, tiling can be applied to explore a better data locality when the size of the matrix is too large to fit into the cache. Such a tiled program is shown in Figure 7, whose matrix size and block size are squares.

```plaintext
 affine BlockMM {n, b| (n,b)>0} given
 float A, B {ii,jj,i,j | 0<=(ii,jj)<n & 0<=(i,j)<b};
 returns
 float C {ii,jj,i,j | 0<=(ii,jj)<n & 0<=(i,j)<b};
 using
 float CC {ii,jj,kk,i,j | 0<=(ii,jj,kk)<n & 0<=(i,j)<b};
 through
 use {ii, jj, kk | 0<=(ii,jj,kk)<n} MatrixMult[b]
 ((ii,jj,kk,i,j->ii,kk,i,j)@A, (ii,jj,kk,i,j->kk,jj,i,j)@B)
 returns (CC); 
 C = reduce(+,
 (ii,jj,kk,i,j->ii,jj,i,j), CC); 
 .
```

Fig. 7. The ALPHA program for tiled matrix multiplication

2) Generated code for tiled matrix multiplication: For the tiled matrix multiplication program, the memory size used for each input/output for all subsystem calls are the same. Therefore, all the subsystem calls can share the same memory allocation for each input/output, i.e., the malloc statements can be hoisted out all the way. Also, the accumulation for the final answer \( C \) can happen right after each of the \( k \)th blocks is computed—the reduction is interspersed with the function calls. The generated with all these optimization is shown in Figure 8.

B. Tiled Forward Substitution

1) Alpha program for tiled forward substitution: The forward substitution program can also be blocked into square tiles
The polyhedral model has recently achieved a significant success in automatic parallelization. It is currently used in research compilers like Loopo [18], Pluto [6] (see http://www.cse.ohio-state.edu/~bondhugu/pluto) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM’s XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of GCC now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

### VI. Related Work

The polyhedral model has recently achieved a significant success in automatic parallelization. It is currently used in research compilers like Loopo [18], Pluto [6] (see http://www.cse.ohio-state.edu/~bondhugu/pluto) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM’s XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of GCC now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

ALPHA [1] is a strongly typed functional language developed by Mauar at IRISA, and extended by Le Verge to include reductions as first class expressions [2], and by Dupont de Dinechin [9], [10] with modular structure. This abstraction has allowed us to develop extremely powerful analyses and transformations such as memory re-allocation [3], detection ofvalue reuse can be enabled for the forward substitution.

The code generated is shown in Figure 10.

### Related Work

The polyhedral model has recently achieved a significant success in automatic parallelization. It is currently used in research compilers like Loopo [18], Pluto [6] (see http://www.cse.ohio-state.edu/~bondhugu/pluto) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM’s XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of GCC now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

ALPHA [1] is a strongly typed functional language developed by Mauar at IRISA, and extended by Le Verge to include reductions as first class expressions [2], and by Dupont de Dinechin [9], [10] with modular structure. This abstraction has allowed us to develop extremely powerful analyses and transformations such as memory re-allocation [3], detection ofvalue reuse can be enabled for the forward substitution.

The code generated is shown in Figure 10.

### Related Work

The polyhedral model has recently achieved a significant success in automatic parallelization. It is currently used in research compilers like Loopo [18], Pluto [6] (see http://www.cse.ohio-state.edu/~bondhugu/pluto) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM’s XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of GCC now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

ALPHA [1] is a strongly typed functional language developed by Mauar at IRISA, and extended by Le Verge to include reductions as first class expressions [2], and by Dupont de Dinechin [9], [10] with modular structure. This abstraction has allowed us to develop extremely powerful analyses and transformations such as memory re-allocation [3], detection ofvalue reuse can be enabled for the forward substitution.

The code generated is shown in Figure 10.

### Related Work

The polyhedral model has recently achieved a significant success in automatic parallelization. It is currently used in research compilers like Loopo [18], Pluto [6] (see http://www.cse.ohio-state.edu/~bondhugu/pluto) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM’s XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of GCC now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

ALPHA [1] is a strongly typed functional language developed by Mauar at IRISA, and extended by Le Verge to include reductions as first class expressions [2], and by Dupont de Dinechin [9], [10] with modular structure. This abstraction has allowed us to develop extremely powerful analyses and transformations such as memory re-allocation [3], detection ofvalue reuse can be enabled for the forward substitution.

The code generated is shown in Figure 10.

### Related Work

The polyhedral model has recently achieved a significant success in automatic parallelization. It is currently used in research compilers like Loopo [18], Pluto [6] (see http://www.cse.ohio-state.edu/~bondhugu/pluto) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM’s XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of GCC now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

ALPHA [1] is a strongly typed functional language developed by Mauar at IRISA, and extended by Le Verge to include reductions as first class expressions [2], and by Dupont de Dinechin [9], [10] with modular structure. This abstraction has allowed us to develop extremely powerful analyses and transformations such as memory re-allocation [3], detection ofvalue reuse can be enabled for the forward substitution.

The code generated is shown in Figure 10.

### Related Work

The polyhedral model has recently achieved a significant success in automatic parallelization. It is currently used in research compilers like Loopo [18], Pluto [6] (see http://www.cse.ohio-state.edu/~bondhugu/pluto) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM’s XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of GCC now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

ALPHA [1] is a strongly typed functional language developed by Mauar at IRISA, and extended by Le Verge to include reductions as first class expressions [2], and by Dupont de Dinechin [9], [10] with modular structure. This abstraction has allowed us to develop extremely powerful analyses and transformations such as memory re-allocation [3], detection ofvalue reuse can be enabled for the forward substitution.

The code generated is shown in Figure 10.
and parallelization of scans and reductions [19] using the algebraic/mathematical properties of semi-ring matrices.

After Dupont de Dinechin’s original work, there was relatively little effort to exploit modularity in ALPHA, partly because the MMALPHA tool did not support automatic tiling, which was at that time an unsolved. It is only with recent work on parametric tiled code generation [20] that modularity in polyhedral equational languages becomes important.

VII. CONCLUSION

The importance of tiling as a program optimization, and the fact that it is in general, a non-linear transformation poses some unique challenges to polyhedral compilation techniques. An approach to overcoming these difficulties is to pre-process the program so that tiling is performed initially, at the specification level. In the polyhedral equational language, ALPHA, subsystems as introduced by Dupont de Dinechin [9], [10] provide us a very natural way of describing such tiled specifications. In this paper, we first presented some new static analyses and transformations for subsystems. We also implemented a code generator that handles subsystems. It generates code with memory and value reuse. The current system requires manual specification of the target mapping although our ongoing work is on automating this.

REFERENCES